**NAMA : FITRIA WIDYA ASTARI**

**NIM : L200150027**

**Tugas-02 :**

**Cari daftar instruction set untuk processor intel dan ARM Android.**

**SET INSTRUKSI ARM ANDROID**

- Advanced RISC Machines (ARM) limited telah mendesain suatu famili mikroprosesor dan melisensikan desain tersebut ke perusahaan lain untuk fabrikasi chip yang penggunaannya dalam produk komputer dan sistem uang embedded.

- Perusahan ARM yang relatif baru, merupakan perkembangan dari perusahan Acorn Computer yang mengembangkan desain prosesor pada awal tahun 1980-an.

- Penggunaan utama mikroprosesor ARM adalah pada aplikasi embedded yang berdaya rendah dan berbiaya rendah, seperti misalnya mobile telephone, modem komunikasi, sistem manejemen mesin mobil, dan hand-held

**3.1 Register, Akses Memori dan Transfer Data**

Dalam arsitektur ARM, memori adalah byte-addressble, menggunakan alamat 32-bit, dan register prosesor memiliki panjang 32-bit. Dua panjang operand digunakan dalam memindahkan data antara memori dan register prosessor : byte (8 bit) dan word (32 bit).

**3.1.1 Struktur Register**

- Register prosesor yang digunakan oleh program aplikasi ditampilkan terdapat enam belas register 32 bit berlabel R0 hingga R15 yang terdiri dari lima belas general-purpose register (R0 hingga R14) dan register program counter (PC), R15, yang terdiri dari 15.

- General purpose register dapat menyimpan alamat memori atau operand data. Current progarm status register (CPRS), atau cukup disebut register status, menyimpan condition code flag (N, Z, C, V), interupt disable flag, dan bit mode prosesor.

- Disini kita akan mengasumsikan bahw prosesor dalam mode user dan dieksekusi sebagai program aplikasi.terdapat 15 general purpose register tambahn yang yang disebut register banked.

**3.1.2 Instruksi Akses Memory Dan Mode Pengalamatan**

- Eksekusi Conditional Instruksi

Fitur yang membedakan dan agak tidak biasa dari prosesor ARM adalah semua instruksinya dieksekusi secara conditional, tergantung pada kondisi yang ditetapkan pada instruksi tersebut. Instruksi tersebut dieksekusi hanya jika keadaan saat ini dari conditional code flag prosesor memenuhi kondisi yang ditetapkan dalam bit b31-28 dari instruksi tersebut. Jika tidak prosesor melanjutkan ke instruksi berikutnya salah satu kondisi tersebut digunakan untuk mengindikasikan bahwa instruksi tersebut selalu dieksekusi

- Mode Pengalamatan Memori

Metode dasar untuk mengalamati operand memori adalah membangkitkan effective address, EA, dari operan tersebut dengan menambahkan offset bertanda keisi base register Rn, yang ditentukan dalam instruksi. Besarnya offset tersebut dapat berupa nilai immediate yang terdapat dalam 12 bit low order instruksi atau isi dari register ketiga, Rm, yang dinamai dengan 4 bit low order tanda arah offset terdapat dalam field OP-code.

- Operand Load/Store Multiple

Selain instruksi load dan store untuk operand tunggal, terdapat 2 instruksi untuk me-load dan menyimpan banyak operand. Instruksi itu disebut instruksi transfer block. Sub set apapun dari general purpose register load atau disimpan. Hanya operand word yang diperbolehkan, dan OP code yang digunakan dalam load multiple dan store multiple. Operand memori harus berada dalam lokasi word yang berurutan.

**3.1.3 Instruksi Move Register**

Acap kali kita perlu meng-copy isi satu register ke register lain atau untuk me-load nilai immediate ke suatu register. Instruksi move

MOV Rd , Rm

**3.2 Instruksi Aritmatika Dan Logika**

Set instruksi ARM memiliki sejumlah instruksi untuk operand aritmatika dan logika pada operand yang berada dalam general-purpose register atau dinyatakan sebagai operand immediate dalam instruksi itu sendiri.

Terdapat instruksi untuk operand logika AND,OR,NOT,XOR, dan bit-clear. Instruksi seperti compare disediakan untuk men-set condition code flag berdasarkan hasil dari operasi aritmatika dan logika pada dua operand

**3.2.1 Instruksi Aritmatika**

Ekspresi bahasa assembly dasar untuk instruksi aritmatika adalah

Opcode Rd, Rn, Rm

Dimana operasi yang ditetapkan oleh OP code dilakukan menggunakan operand dalam general-purpose register Rn dan Rm. Hasilnya diletakkan dalam register Rd. Misalnya, instruksi

ADD R0, R2, R4

Menjalankan operasi

R0 🡨 [R2] + [R4]

Dan instruksi

SUB R0, R6, R5

Menjalankan operasi

R0 🡨 [R6] – [R5]

**3.2.2 Instruksi Logika**

Operasi logika AND, OR, XOR, dan Bit-clear diimplementasikan oleh instruksi OP code AND, ORR, EOR, dan BIC. Kode tersebut memiliki format yang sama dengan instruksi aritmatika. Instruksi

AND Rd, Rn, Rm

Menjalankan operasi

Rd 🡨 [Rn] ^ [Rm]

Yang merupakan bitwise logical AND antara operand dalam register Rn dan Rm. Misalnya, jika register R0 berisi pola hexadesimal 02FA62CA dan R1 berisi pola 0000FFFF, maka instruksi

AND R0, RO, R1

Akan menyebabkan pola 000062CA diletakkan dalam register R0.

**3.3 Instruksi Branch**

Instruksi branch CONDITIONAL berisi offset 24-bit, 2’-complement, bertanda yang ditambahkan ke isi ter-update Program Counter untuk menghasilkan alamat target branch.

Instruksi Branch dieksekusi dengan cara yang sama seperti instruksi ARM yang lain, yaitu dieksekusi hanya jika keadaan terbaru condition code flag berhubungan dengan kondisi ditetapkan dalam field codition instruksi tersebut.

**3.3.1 Setting Condition Code**

beberapa instruksi, seperti compare, dinyatakan sebagai berikut

CMP Rn, Rm

Yang menjalankan operasi

[Rn] – [Rm]

Memiliki tujuan utama untuk men-set condition code flag berdasar pada hasil operasi pengurangan.

**3.3.2 Program Loop untuk penambahan bilangan**

Operasi load dan store dilakukan oleh instruksi pertama, kedua, dan terakhir yang digunakan oleh mode pengalamatan relative. Ini mengasumsikan bahwa lokasi memori N, pointer, dan SUM terdapat dalam rentang yang terjangkau oleh offset relatif terhadap PC. Lokasi memori pointer berisi alamat NUM 1 dari bilangan pertama yang akan ditambahkan, N berisi jumlah entri didalam list dan SUM digunakan untuk menyimpan jumlah tersebut

**3.4 Bahasa Assembly**

Bahasa assembly ARM memiliki assembler directive untuk menyiapkan ruang penyimpanan, menetapkan nilai numerik ke label alamat dari simbol konstanta, menentukan dimana program dan blok data akan ditempatkan dalam memori, menetapkan akhir teks source program fasilitas tersebut didekskripsikan secara umum.

**3.5 Kontrol Aliran Program**

- Condition Code Flag

68000 memiliki lima condition code flag, disimpan dalam register status. Selain flag M,Z,V, dan C yang dideskripsikan pada bagian 2.4.6, 6800 memiliki lima flag, X (extend). Di-set dengan cara yang sama dengan flag C, tetapi tidak dipengaruhi oleh banyak instruksi.

- Instruksi Branch

Instruksi conditional branch menyebabkan eksekusi program berlanjut dengan instruksi pada alamat target branch jika kondisi branch dipenuhi.

**3.6 Operasi I/O**

Prosessor 68000 mensyaratkan semua status dan buffer data dalam antar muka perangkat I/O menjadi addressable seperti lokasi memori. Ini berarti program-program I/O dalam komputer 68000 dapat dicapai.

**3.7 Stack dan Subroutine**

Suatu stack dapat diimplementasikan, menggunakan register alamat apapun sebagai pointer. Mode pengalamatan Autoincrement dan Autodecrement memfasilitasi proses ini. Satu register spesifik, register A7, ditunjuk sebagai pointer stack prosesor, dan stack yang ditunjuk register ini disebut stack prosesor. Ini adalah stack yang digunakan dalam semua operasi stack yang dilakukan prosesor secara otomatis, sebagaimana dalam kasus linkage subroutine.

**3.8 Instruksi Logika**

Operand yang terlibat dalam instruksi ini memiliki panjang tetap 32, 16, atau 8 bit. Pada beberapa aplikasi perlu untuk memanipulasi: ukuran data yang lain, mungkin hanya bit individu, dan melakukan operasi logika pada data ini.

**3.9 Register dan Pengalamatan**

Dalam arsitektur 1A-32, memori adalah byte addressable menggunakan alamat 32-bit, dan instruksi beroperasi pada operand data 8 dan 32 bit. Ukuran operand ini disebut byte dan doubleword dalam istilah intel.

**3.9.1 Struktur Register 1A-32**

Terdapat delapan floating-point register untuk menyimpan operand data floating point doubleword atau quadword (64 bit). Floating-point register yang memiliki field ekstensi untuk menyediakan panjang total 80 bit.

Arsitektur 1A-32 berbasis pada model memori yang menghubungkan area yang berbeda di dalam memori, yang disebut segmen dengan kegunaan yang berbeda.

**3.9.2 Mode Pengalamatan 1A-32**

- Arsitektur 1A-32 memiliki set mode pengalamatan yang besar dan fleksibel. Mode tersebut didesain untuk mengakses item data individu atau item data yang merupakan aggota dari list yang berurutan yang mulai pada alamat memori tertentu.

- Mode dasar, yang tersedia pada kebanyakan prosesor telah dideskripsikan. Mode tersebut adalah : Immediate, Absolute, Register, dan Register indirect.

**INSTRUCTION SET LIST INTEL**

This section lists all the Intel Architecture instructions divided into three major groups: integer, MMX technology, floating-point, and system instructions. For each instruction, the mnemonic and descriptive names are given. When two or more mnemonics are given (for example, CMOVA/CMOVNBE), they represent different mnemonics for the same instruction opcode. Assemblers support redundant mnemonics for some instructions to make it easier to read code listings. For instance, CMOVA (Conditional move if above) and CMOVNBE (Conditional move is not below or equal) represent the same condition.

**30.2.1 Integer Instructions**

Integer instructions perform the integer arithmetic, logic, and program flow control operations that programmers commonly use to write application and system software to run on an Intel Architecture processor. In the following sections, the integer instructions are divided into several instruction subgroups.

**30.2.1.1 Data Transfer Instructions**

MOV Move

CMOVE/CMOVZ Conditional move if equal/Conditional move if zero

CMOVNE/CMOVNZ Conditional move if not equal/Conditional move if not zero

CMOVA/CMOVNBE Conditional move if above/Conditional move if not below or equal

CMOVAE/CMOVNB Conditional move if above or equal/Conditional move if not below

CMOVB/CMOVNAE Conditional move if below/Conditional move if not above or equal

CMOVBE/CMOVNA Conditional move if below or equal/Conditional move if not above

CMOVG/CMOVNLE Conditional move if greater/Conditional move if not less or equal

CMOVGE/CMOVNL Conditional move if greater or equal/Conditional move if not less

CMOVL/CMOVNGE Conditional move if less/Conditional move if not greater or equal

CMOVLE/CMOVNG Conditional move if less or equal/Conditional move if not greater

CMOVC Conditional move if carry

CMOVNC Conditional move if not carry

CMOVO Conditional move if overflow

CMOVNO Conditional move if not overflow

CMOVS Conditional move if sign (negative)

CMOVNS Conditional move if not sign (non-negative)

CMOVP/CMOVPE Conditional move if parity/Conditional move if parity even

CMOVNP/CMOVPO Conditional move if not parity/Conditional move if parity odd

XCHG Exchange

BSWAP Byte swap

XADD Exchange and add

CMPXCHG Compare and exchange

CMPXCHG8B Compare and exchange 8 bytes

PUSH Push onto stack

POP Pop off of stack

PUSHA/PUSHAD Push general-purpose registers onto stack

POPA/POPAD Pop general-purpose registers from stack

IN Read from a port

OUT Write to a port

CWD/CDQ Convert word to doubleword/Convert doubleword to quadword

CBW/CWDE Convert byte to word/Convert word to doubleword in EAX register

MOVSX Move and sign extend

MOVZX Move and zero extend

**30.2.1.2 Binary Arithmetic Instructions**

ADD Integer add

ADC Add with carry

SUB Subtract

SBB Subtract with borrow

IMUL Signed multiply

MUL Unsigned multiply

IDIV Signed divide

DIV Unsigned divide

INC Increment

DEC Decrement

NEG Negate

CMP Compare

**30.2.1.3 Decimal Arithmetic**

DAA Decimal adjust after addition

DAS Decimal adjust after subtraction

AAA ASCII adjust after addition

AAS ASCII adjust after subtraction

AAM ASCII adjust after multiplication

AAD ASCII adjust before division

**30.2.1.4 Logic Instructions**

AND And

OR Or

XOR Exclusive or

NOT Not

**30.2.1.5 Shift and Rotate Instructions**

SAR Shift arithmetic right

SHR Shift logical right

SAL/SHL Shift arithmetic left/Shift logical left

SHRD Shift right double

SHLD Shift left double

ROR Rotate right

ROL Rotate left

RCR Rotate through carry right

RCL Rotate through carry left

**30.2.1.6 Bit and Byte Instructions**

BT Bit test

BTS Bit test and set

BTR Bit test and reset

BTC Bit test and complement

BSF Bit scan forward

BSR Bit scan reverse

SETE/SETZ Set byte if equal/Set byte if zero

SETNE/SETNZ Set byte if not equal/Set byte if not zero

SETA/SETNBE Set byte if above/Set byte if not below or equal

SETAE/SETNB/SETNC Set byte if above or equal/Set byte if not below/Set byte if not carry

SETB/SETNAE/SETC Set byte if below/Set byte if not above or equal/Set byte if carry

SETBE/SETNA Set byte if below or equal/Set byte if not above

SETG/SETNLE Set byte if greater/Set byte if not less or equal

SETGE/SETNL Set byte if greater or equal/Set byte if not less

SETL/SETNGE Set byte if less/Set byte if not greater or equal

SETLE/SETNG Set byte if less or equal/Set byte if not greater

SETS Set byte if sign (negative)

SETNS Set byte if not sign (non-negative)

SETO Set byte if overflow

SETNO Set byte if not overflow

SETPE/SETP Set byte if parity even/Set byte if parity

SETPO/SETNP Set byte if parity odd/Set byte if not parity

TEST Logical compare

**30.2.1.7 Control Transfer Instructions**

JMP Jump

JE/JZ Jump if equal/Jump if zero

JNE/JNZ Jump if not equal/Jump if not zero

JA/JNBE Jump if above/Jump if not below or equal

JAE/JNB Jump if above or equal/Jump if not below

JB/JNAE Jump if below/Jump if not above or equal

JBE/JNA Jump if below or equal/Jump if not above

JG/JNLE Jump if greater/Jump if not less or equal

JGE/JNL Jump if greater or equal/Jump if not less

JL/JNGE Jump if less/Jump if not greater or equal

JLE/JNG Jump if less or equal/Jump if not greater

JC Jump if carry

JNC Jump if not carry

JO Jump if overflow

JNO Jump if not overflow

JS Jump if sign (negative)

JNS Jump if not sign (non-negative)

JPO/JNP Jump if parity odd/Jump if not parity

JPE/JP Jump if parity even/Jump if parity

JCXZ/JECXZ Jump register CX zero/Jump register ECX zero

LOOP Loop with ECX counter

LOOPZ/LOOPE Loop with ECX and zero/Loop with ECX and equal

LOOPNZ/LOOPNE Loop with ECX and not zero/Loop with ECX and not equal

CALL Call procedure

RET Return

IRET Return from interrupt

INT Software interrupt

INTO Interrupt on overflow

BOUND Detect value out of range

ENTER High-level procedure entry

LEAVE High-level procedure exit

**30.2.1.8 String Instructions**

MOVS/MOVSB Move string/Move byte string

MOVS/MOVSW Move string/Move word string

MOVS/MOVSD Move string/Move doubleword string

CMPS/CMPSB Compare string/Compare byte string

CMPS/CMPSW Compare string/Compare word string

CMPS/CMPSD Compare string/Compare doubleword string

SCAS/SCASB Scan string/Scan byte string

SCAS/SCASW Scan string/Scan word string

SCAS/SCASD Scan string/Scan doubleword string

LODS/LODSB Load string/Load byte string

LODS/LODSW Load string/Load word string

LODS/LODSD Load string/Load doubleword string

STOS/STOSB Store string/Store byte string

STOS/STOSW Store string/Store word string

STOS/STOSD Store string/Store doubleword string

REP Repeat while ECX not zero

REPE/REPZ Repeat while equal/Repeat while zero

REPNE/REPNZ Repeat while not equal/Repeat while not zero

INS/INSB Input string from port/Input byte string from port

INS/INSW Input string from port/Input word string from port

INS/INSD Input string from port/Input doubleword string from port

OUTS/OUTSB Output string to port/Output byte string to port

OUTS/OUTSW Output string to port/Output word string to port

OUTS/OUTSD Output string to port/Output doubleword string to port

**I/O INSTRUCTIONS**

The IN (input from port to register), INS (input from port to string), OUT (output from register to port), and OUTS (output string to port) instructions move data between the processor’s I/O ports and either a register or memory.

The register I/O instructions (IN and OUT) move data between an I/O port and the EAX register (32-bit I/O), the AX register (16-bit I/O), or the AL (8-bit I/O) register. The I/O port being read or written to is specified with an immediate operand or an address in the DX register.

The block I/O instructions (INS and OUTS) instructions move blocks of data (strings) between an I/O port and memory. These instructions operate similar to the string instructions (see “String Operations”). The ESI and EDI registers are used to specify string elements in memory and the repeat prefixes (REP) are used to repeat the instructions to implement block moves. The assembler recognizes the following alternate mnemonics for these instructions: INSB (input byte), INSW (input word), and INSD (input doubleword), and OUTB (output byte), OUTW (output word), and OUTD (output doubleword).

The INS and OUTS instructions use an address in the DX register to specify the I/O port to be read or written to.

**JUMP INSTRUCTION**

The JMP (jump) instruction unconditionally transfers program control to a destination instruction. The transfer is one-way; that is, a return address is not saved. A destination operand specifies the address (the instruction pointer) of the destination instruction. The address can be a relative addressor an absolute address.

A relative address is a displacement (offset) with respect to the address in the EIP register. The destination address (a near pointer) is formed by adding the displacement to the address in the EIP register. The displacement is specified with a signed integer, allowing jumps either forward or backward in the instruction stream.

An absolute address is a offset from address 0 of a segment. It can be specified in either of the following ways:

• An address in a general-purpose register.This address is treated as a near pointer, which is

copied into the EIP register. Program execution then continues at the new address within the

current code segment.

• An address specified using the standard addressing modes of the processor.Here, the

address can be a near pointer or a far pointer. If the address is for a near pointer, the address is translated into an offset and copied into the EIP register. If the address is for a far pointer, the address is translated into a segment selector (which is copied into the CS register) and an offset (which is copied into the EIP register).

In protected mode, the JMP instruction also allows jumps to a call gate, a task gate, and a task-state segment.

**CONDITIONAL JUMP INSTRUCTIONS**

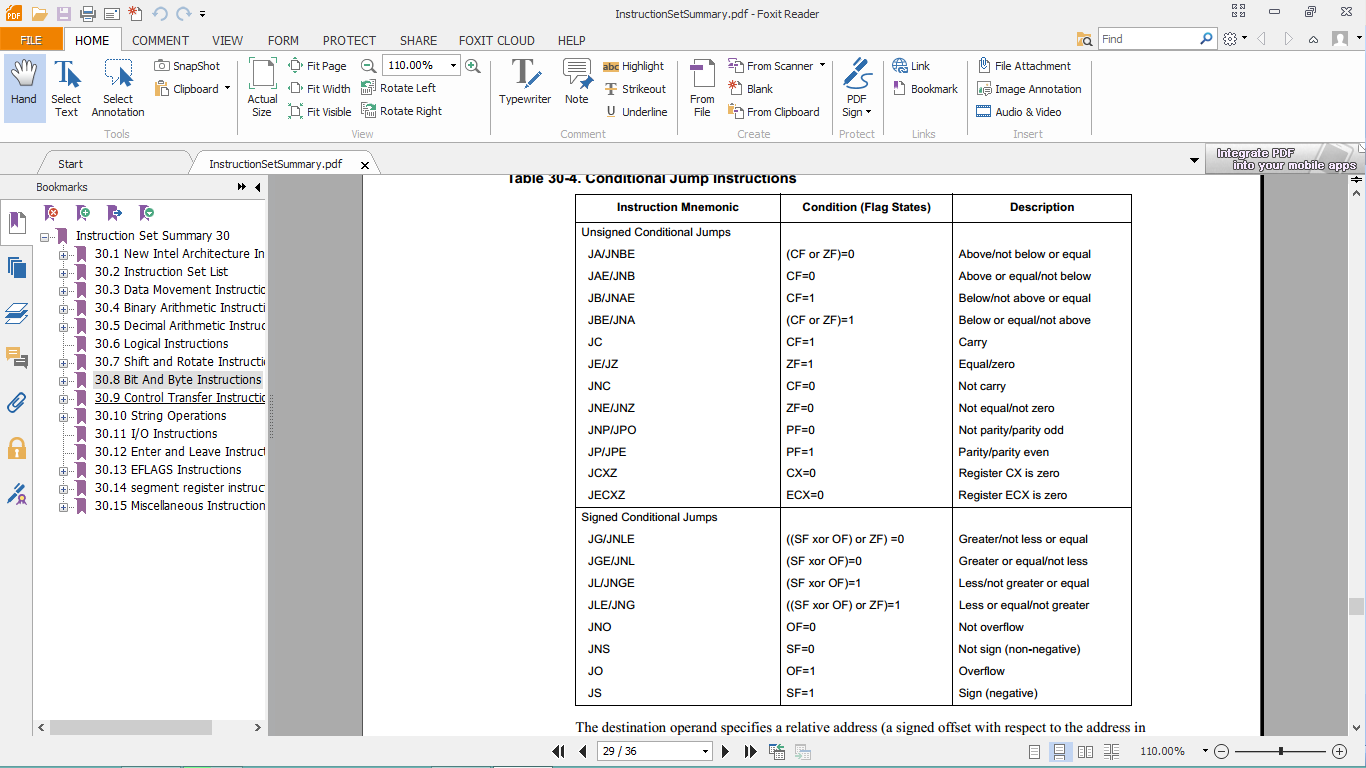
The Jcc(conditional) jump instructions transfer program control to a destination instruction if the conditions specified with the condition code (cc) associated with the instruction are satisfied (see Table 30-4). If the condition is not satisfied, execution continues with the instruction following the Jccinstruction. As with the JMP instruction, the transfer is one-way; that is, a return address is not saved.

The destination operand specifies a relative address (a signed offset with respect to the address in the EIP register) that points to an instruction in the current code segment. The Jccinstructions do not support far transfers; however, far transfers can be accomplished with a combination of a Jccand a JMP instruction (see “Jcc—Jump if Condition Is Met” in Chapter 3 of the Intel Architecture Software Developer’s Manual, Volume 2).Table 30-4 shows the mnemonics for the Jccinstructions and the conditions being tested for each instruction. The condition code mnemonics are appended to the letter “J” to form the mnemonic for a Jccinstruction. The instructions are divided into two groups: unsigned and signed conditional

jumps. These groups correspond to the results of operations performed on unsigned and signed integers, respectively. Those instructions listed as pairs (for example, JA/JNBE) are alternate names for the same instruction. The assembler provides these alternate names to make it easier to read program listings.

The JCXZ and JECXZ instructions test the CX and ECX registers, respectively, instead of one or more status flags. See “Jump If Zero Instructions” for more information about these instructions.

**Table 30-4. Conditional Jump Instructions**

****

The destination operand specifies a relative address (a signed offset with respect to the address in the EIP register) that points to an instruction in the current code segment. The Jccinstructions do not support far transfers; however, far transfers can be accomplished with a combination of a Jccand a JMP instruction (see “Jcc—Jump if Condition Is Met” in Chapter 3 of the Intel Architecture Software Developer’s Manual, Volume 2).

Table 30-4 shows the mnemonics for the Jccinstructions and the conditions being tested for each instruction. The condition code mnemonics are appended to the letter “J” to form the mnemonic for a Jccinstruction. The instructions are divided into two groups: unsigned and signed conditional

jumps. These groups correspond to the results of operations performed on unsigned and signed integers, respectively. Those instructions listed as pairs (for example, JA/JNBE) are alternate names for the same instruction. The assembler provides these alternate names to make it easier to read program listings.

The JCXZ and JECXZ instructions test the CX and ECX registers, respectively, instead of one or more status flags. See “Jump If Zero Instructions” for more information about these instructions.

**30.9.2.2 Loop Instructions**

The LOOP, LOOPE (loop while equal), LOOPZ (loop while zero), LOOPNE (loop while not

equal), and LOOPNZ (loop while not zero) instructions are conditional jump instructions that use the value of the ECX register as a count for the number of times to execute a loop. All the loop instructions decrement the count in the ECX register each time they are executed and terminate a loop when zero is reached. The LOOPE, LOOPZ, LOOPNE, and LOOPNZ instructions also accept the ZF flag as a condition for terminating the loop before the count reaches zero.

The LOOP instruction decrements the contents of the ECX register (or the CX register, if the

address-size attribute is 16), then tests the register for the loop-termination condition. If the count in the ECX register is non-zero, program control is transferred to the instruction address specified by the destination operand. The destination operand is a relative address (that is, an offset relative to the contents of the EIP register), and it generally points to the first instruction in the block of code that is to be executed in the loop. When the count in the ECX register reaches zero, program control is transferred to the instruction immediately following the LOOP instruction, which terminates the loop. If the count in the ECX register is zero when the LOOP instruction is first executed, the register is pre-decremented to FFFFFFFFH, causing the loop to be executed 232 times.

The LOOPE and LOOPZ instructions perform the same operation (they are mnemonics for the same instruction). These instructions operate the same as the LOOP instruction, except that they also test the ZF flag. If the count in the ECX register is not zero and the ZF flag is set, program control is transferred to the destination operand. When the count reaches zero or the ZF flag is clear, the loop is terminated by transferring program control to the instruction immediately following the LOOPE/LOOPZ instruction.

The LOOPNE and LOOPNZ instructions (mnemonics for the same instruction) operate the same as the LOOPE/LOOPPZ instructions, except that they terminate the loop if the ZF flag is set.

**30.9.2.3 Jump If Zero Instructions**

The JECXZ (jump if ECX zero) instruction jumps to the location specified in the destination

operand if the ECX register contains the value zero. This instruction can be used in combination with a loop instruction (LOOP, LOOPE, LOOPZ, LOOPNE, or LOOPNZ) to test the ECX register prior to beginning a loop. As described in “Loop Instructions”, the loop instructions decrement the contents of the ECX register before testing for zero. If the value in the ECX register is zero initially, it will be decremented to FFFFFFFFH on the first loop instruction, causing the loop to be executed 232times. To prevent this problem, a JECXZ instruction can be inserted at the beginning

of the code block for the loop, causing a jump out the loop if the EAX register count is initially zero. When used with repeated string scan and compare instructions, the JECXZ instruction can determine whether the loop terminated because the count reached zero or because the scan or compare conditions were satisfied.

The JCXZ (jump if CX is zero) instruction operates the same as the JECXZ instruction when the 16-bit address-size attribute is used. Here, the CX register is tested for zero.